Distance Range Queries in SpatialHadoop

Francisco García-García[1,⋆], Antonio Corral[1,⋆], Luis Iribarne[1,⋆], and Michael Vassilakopoulos[2,⋆]

[1] Dept. of Informatics, University of Almeria, Almeria, Spain. E-mail: {paco.garcia,acorral,liribarn}@ual.es
[2] Dept. of Electrical and Computer Engineering, University of Thessaly, Volos, Greece. E-mail: mvasilako@uth.gr

Abstract. Efficient processing of Distance Range Queries (*DRQs*) is of great importance in spatial databases due to the wide area of applications. This type of spatial query is characterized by a *distance range* over one or two datasets. The most representative and known *DRQs* are the ε Distance Range Query (εDRQ) and the ε Distance Range Join Query ($\varepsilon DRJQ$). Given the increasing volume of spatial data, it is difficult to perform a DRQ on a centralized machine efficiently. Moreover, the $\varepsilon DRJQ$ is an expensive spatial operation, since it can be considered a combination of the εDR and the spatial join queries. For this reason, this paper addresses the problem of computing *DRQs* on big spatial datasets in SpatialHadoop, an extension of Hadoop that supports spatial operations efficiently, and proposes new algorithms in SpatialHadoop to perform efficient parallel *DRQs* on large-scale spatial datasets. We have evaluated the performance of the proposed algorithms in several situations with big synthetic and real-world datasets. The experiments have demonstrated the efficiency and scalability of our proposal.

Keywords: Distance Range Queries, Spatial Data Processing, SpatialHadoop, MapReduce

1 Introduction

Distance Range Queries (*DRQs*) have received considerable attention from the database community, due to its importance in numerous applications, such as spatial databases and GIS [1]. For the ε Distance Range Query (εDRQ), given one point dataset P, one query point q and a distance range $[\varepsilon_1, \varepsilon_2]$, this spatial query finds all points of P that are contained inside a region of circular shape or *annulus* centered in q with radios ε_1 and ε_2 (i.e. the region in the plane between two concentric circles of different radius). For the case of the ε Distance Range Join Query ($\varepsilon DRJQ$), given two point datasets P and Q and a distance range $[\varepsilon_1, \varepsilon_2]$, this distance-based join query finds all the possible pairs of points from $P \times Q$, having a distance between ε_1 and ε_2 of each other. Lots of researches

⋆ Work funded by the MINECO research project [TIN2013-41576-R] and the Junta de Andalucia research project [P10-TIC-6114] have worked on the improvement of the performance of $DRQs$ by proposing efficient algorithms or designing new complex spatial index structures. However, all these approaches focus on methods that are to be executed in a centralized environment.

With the fast increase in the scale of big input datasets, processing large data in parallel and distributed fashions is becoming a popular practice. A number of parallel algorithms for Spatial Join [2, 3], K Nearest Neighbor (KNN) Join [4, 5], K Closest Pair Query (KCPQ) [6], top-K Similarity Join [7] and Similarity Join [8] in MapReduce [9] have been designed and implemented recently. But, to the authors' knowledge, there is no research works on parallel and distributed $DRQs$ (εDRQ and εDJQ) in large spatial data, which is a challenging task and becoming increasingly essential as datasets continue growing.

Actually, parallel and distributed computing using shared-nothing clusters on extreme-scale data is becoming a dominating trend in the context of data processing and analysis. MapReduce [9] is a framework for processing and managing large-scale datasets in a distributed cluster, which has been used for applications such as generating search indexes, document clustering, access log analysis, and various other forms of data analysis [10]. MapReduce was introduced with the goal of providing a simple yet powerful parallel and distributed computing paradigm, providing good scalability and fault tolerance mechanisms. The success of MapReduce stems from hiding the details of parallelization, fault tolerance, and load balancing in a simple programming framework.

However, as also indicated in [11], MapReduce has weaknesses related to efficiency when it needs to be applied to spatial data. A main shortcoming is the lack of any indexing mechanism that would allow selective access to specific regions of spatial data, which would in turn yield more efficient query processing algorithms. A recent solution to this problem is an extension of Hadoop, called SpatialHadoop [12], which is a framework that inherently supports spatial indexing on top of Hadoop. In SpatialHadoop, spatial data is deliberately partitioned and distributed to nodes, so that data with spatial proximity is placed in the same partition. Moreover, the generated partitions are indexed, thereby enabling the design of efficient query processing algorithms that access only part of the data and still return the correct result query. As demonstrated in [12], various algorithms are proposed for spatial queries, such as range, KNN, spatial joins, skyline[13] and KCP [6] queries. Efficient processing of $DRQs$ (εDRQ and εDJQ) over large-scale spatial datasets is a challenging task, and it is the main target of this paper.

Motivated by these observations, we first propose new parallel algorithms, based on plane-sweep technique, for the εDRQ and $\varepsilon DRJQ$ in SpatialHadoop on big spatial datasets. And next, we present the execution of a set of experiments that demonstrate the efficiency and scalability of our proposal using big synthetic and real-world points datasets.

This paper is organized as follows. In Section 2 we review related work on Hadoop systems that support spatial operations, the specific spatial queries using MapReduce and provide the motivation for this paper. In Section 3, we present preliminary concepts related to $DRQs$ and SpatialHadoop. In section 4 the parallel algorithms for processing εDRQ and $\varepsilon DRJQ$ in SpatialHadoop are proposed. In Section 5, we present representative results of the extensive experimentation that we have performed, using real-world and synthetic datasets, for comparing the efficiency of the proposed algorithms, taking into account different performance parameters. Finally, in Section 6 we provide the conclusions arising from our work and discuss related future work directions.

2 Related Work and Motivation

Researchers, developers and practitioners worldwide have started to take advantage of the MapReduce environment in supporting large-scale data processing. The most important contributions in the context of scalable spatial data processing are the following prototypes: (1) *Parallel-Secondo* [14] is a parallel spatial DBMS that uses Hadoop as a distributed task scheduler; (2) *Hadoop-GIS* [15] extends Hive [16], a data warehouse infrastructure built on top of Hadoop with a uniform grid index for range queries, spatial joins and other spatial operations. It adopts Hadoop Streaming framework and integrates several open source software packages for spatial indexing and geometry computation; (3) *SpatialHadoop* [12] is a full-fledged MapReduce framework with native support for spatial data. It tightly integrates well-known spatial operations (including indexing and joins) into Hadoop; (4) *SpatialSpark* [17] is a lightweight implementation of several spatial operations on top of the *Apache Spark*[1] in-memory big data system. It targets at in-memory processing for higher performance; and (5) *GeoSpark* [18] is an in-memory cluster computing system for processing large-scale spatial data, and it extends the core of *Apache Spark* to support spatial data types, indexes, and operations. It is important to highlight that these five systems differ significantly in terms of distributed computing platforms, data access models, programming languages and the underlying computational geometry libraries.

Actually, there are several works on specific spatial queries using MapReduce. This programming framework adopts a flexible computation model with a simple interface consisting of *map* and *reduce* functions whose implementations can be customized by application developers. Therefore, the main idea is to develop *map* and *reduce* functions for the required spatial operation, which will be executed on-top of an existing Hadoop cluster. Examples of such works on specific spatial queries include: (1) *Region query* [19, 20], where the input file is scanned, and each record is compared against the query range. (2) *KNN query* [20, 21], where a brute force approach calculates the distance to each point and selects the nearest K points [20], while another approach partitions points using a Voronoi diagram and finds the answer in partitions close to the query point [21]. (3) *Skyline query* [13, 24, 25]; in [24] the authors propose algorithms for processing skyline and reverse skyline queries in MapReduce; and in [13, 25] the problem of computing the skyline of a vast-sized spatial dataset in SpatialHadoop is studied. (4) *Reverse Nearest Neighbor (RNN) query* [21], where input data is partitioned

[1] `http://spark.apache.org/`

by a Voronoi diagram to exploit its properties to answer RNN queries. (5) *Spatial join* [12, 20, 22]; in [20] the *partition-based spatial-merge* join [23] is ported to MapReduce, and in [12] the *map* function partitions the data using a grid while the *reduce* function joins data in each grid cell. (6) *KNN join* [4, 5, 22], where the main target is to find for each point in a set P, its KNN points from set Q using MapReduce. (7) in [7], the problem of the *top-K closest pair problem* (where just one dataset is involved) with Euclidean distance using MapReduce is studied. (8) *KCP query* [6], the problem of answering the KCPQ (using plane-sweep techniques [28]) in SpatialHadoop is studied and evaluated. Finally, (9) the *similarity join* in high-dimensional data using MapReduce has been actively studied, the most representative work is [8], where a partition-based similarity join for MapReduce is proposed (called *MRSimJoin*). This approach is based on the *QuickJoin* algorithm [26], and iteratively partitions the data until each partition can be processed on a single reducer (i.e. it partitions and distributes the data until the subsets are small enough to be processed in a single node).

DRQs have received considerable attention from the spatial database community, due to its importance in numerous applications [1]. SpatialHadoop is equipped with several spatial operations, including window query, KNN and spatial join [12], and other fundamental computational geometry algorithms as polygon union, skyline, convex hull, farthest pair, and closest pair [28]. And recently, new algorithms for skyline query processing have been also proposed in [13] and for KCPQ in [6]. And based on the previous observations, efficient processing of *DRQs* over large-scale spatial datasets using SpatialHadoop is a challenging task, and it is the main motivation of this paper.

3 Preliminaries and Background

In this section, we first present the basic definitions of the *DRQs*, followed by a brief introduction of preliminary concepts of SpatialHadoop, which is a full-fledged MapReduce framework with native support for spatial data.

3.1 Distance Range Queries

εDistance Range Query The ε Distance Range Query (εDRQ) reports all spatial objects from a spatial objects dataset that fall on the distance range defined by $[\varepsilon_1, \varepsilon_2]$ with respect to a query object. That is, it finds all spatial objects form the spatial dataset that are contained with the *annulus* centered in the query point with radios ε_1 and ε_2 (i.e. the region in the plane between two concentric circles of different radius). It is a special case of *Regional Query* [1], which permits search regions to have arbitrary orientations and shape. In our case, the region query is defined by a point query and an interval of distances, generating different circular shapes (e.g. if $\varepsilon_1 = 0$ and $\varepsilon_2 > 0$, then the region is a circle; if $\varepsilon_1 = \varepsilon_2 > 0$, then the region is a circumference; if $\varepsilon_1 = \varepsilon_2 = 0$, then the spatial objects intersect or they can be identical; etc.). The formal definition of εDRQ for point datasets (the extension of this definition to other complex spatial objects is straightforward) is the following:

Definition 1. (*ε* **Distance Range Query,** *εDRQ*)
Let $P = \{p_0, p_1, \cdots, p_{n-1}\}$ a set of points in E^d, a query point q in E^d, and a distance range defined by $[\varepsilon_1, \varepsilon_2]$ such that ε_1, $\varepsilon_2 \in \mathbb{R}^+$ and $\varepsilon_1 \leq \varepsilon_2$. Then, the result of the ε Distance Range Query with respect to the query point q is a set $\varepsilon DRQ(P, q, \varepsilon_1, \varepsilon_2) \subseteq P$, which contains all points $p_i \in P$ that fall on the circular shape, centered in q with radios ε_1 and ε_2:

$$\varepsilon DRQ(P, q, \varepsilon_1, \varepsilon_2) = \{p_i \in P \ : \ \varepsilon_1 \leq \ dist(p_i, q) \ \leq \ \varepsilon_2\}$$

An example of this spatial query could be to find all fitness centers between 1 and 2 kilometers from my home. This spatial query has been studied in centralized environments, however, when the dataset resides in a parallel and distributed framework, it has not been given enough attention.

εDistance Range Join Query The ε Distance Range Join Query ($\varepsilon DRJQ$) reports all the possible pairs of spatial objects from two different spatial objects datasets, having a distance between ε_1 and ε_2 of each other. $\varepsilon DRJQ$ is a generalization of the ε Distance Join Query (εDJQ), which is characterized by two spatial datasets and a distance threshold, ε, and it permits search pairs of spatial objects from two input datasets that are within distance ε from each other. In our case, the distance threshold is a range defined by an interval of distances $[\varepsilon_1, \varepsilon_2]$, and if $\varepsilon_1 = 0$ and $\varepsilon_2 > 0$, then we have the definition of εDJQ. Moreover, if $\varepsilon_1 = \varepsilon_2 = 0$, then we have the condition of Spatial Intersection Join, which retrieves all different intersecting spatial object pairs from two distinct spatial datasets [1]. This query is also related to the Similarity Join [8], where the problem of deciding if two objects are similar is reduced to the problem of determining if two high-dimensional points are within a certain distance threshold ε of each other.

Definition 2. (*ε* **Distance Range Join Query,** *$\varepsilon DRJQ$*)
Let $P = \{p_0, p_1, \cdots, p_{n-1}\}$ and $Q = \{q_0, q_1, \cdots, q_{m-1}\}$ be two set of points in E^d, and a distance range defined by $[\varepsilon_1, \varepsilon_2]$ such that ε_1, $\varepsilon_2 \in \mathbb{R}^+$ and $\varepsilon_1 \leq \varepsilon_2$. Then, the result of the ε Distance Range Join Query is a set $\varepsilon DRJQ(P, Q, \varepsilon_1, \varepsilon_2) \subseteq P \times Q$, which contains all the possible different pairs of points that can be formed by choosing one point $p_i \in P$ and one point $q_j \in Q$, having a distance between ε_1 and ε_2 of each other:

$$\varepsilon DRJQ(P, Q, \varepsilon_1, \varepsilon_2) = \{(p_i, q_j) \in P \times Q \ : \ \varepsilon_1 \leq \ dist(p_i, q_j) \ \leq \ \varepsilon_2\}$$

An example scenario of this spatial distance-based join query could be the following. Suppose we want to buy a house with convenient living surroundings. We find a collection of possible houses P and a collection of shopping centers as Q, with the units of x and y axis being in kilometers (km). Suppose that we wants to consider only houses with their distances to shopping centers being at most 1500 meters. So we issues an $\varepsilon DRJQ$ with $\varepsilon_1 = 0km$ and $\varepsilon_2 = 1.5km$. As the εDRQ, this spatial query has been also studied in centralized environments, but when the datasets reside in a parallel and distributed framework, it has not attracted similar attention. For this reason, the main target of this work is to study these spatial query in SpatialHadoop.

3.2 SpatialHadoop

SpatialHadoop [12] is a full-fledged MapReduce framework with native support for spatial data. Notice that MapReduce [11] is a scalable, flexible and fault-tolerant programming framework for distributed large-scale data analysis. A task to be performed using the MapReduce framework has to be specified as two phases: the *Map* phase is specified by a *map function* takes input (typically from Hadoop Distributed File System (HDFS) files), possibly performs some computations on this input, and distributes it to worker nodes; and the *Reduce* phase which processes these results as specified by a *reduce function*. An important aspect of MapReduce is that both the input and the output of the *Map* step are represented as *Key/Value pairs*, and that pairs with same key will be processed as one group by the *Reducer*: $map : (k_1, v_1) \rightarrow list(k_2, v_2)$ and $reduce : k_2, list(v_2) \rightarrow list(v_3)$. Additionally, a *Combiner function* can be used to run on the output of *Map* phase and perform some filtering or aggregation to reduce the number of keys passed to the *Reducer*.

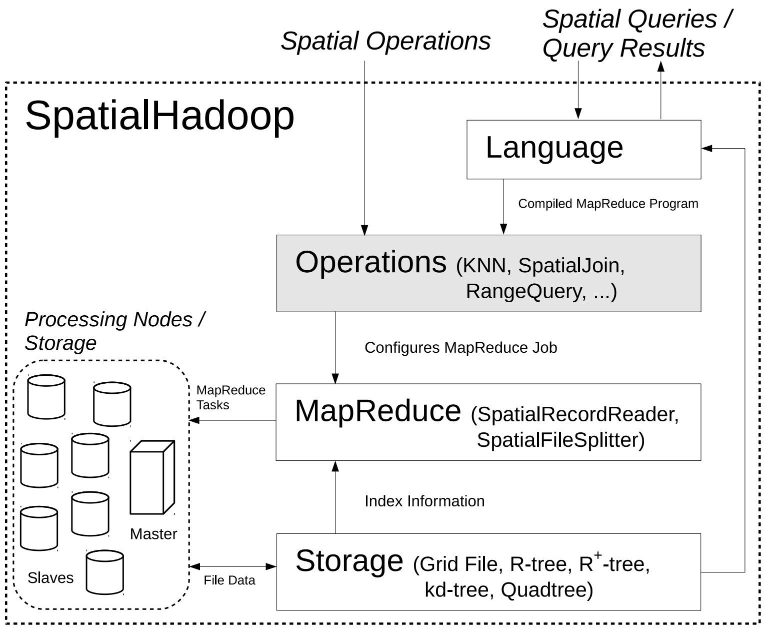
Spatial Operations
Spatial Queries / Query Results
SpatialHadoop
Language
Compiled MapReduce Program
Operations (KNN, SpatialJoin, RangeQuery, ...)
Processing Nodes / Storage
Configures MapReduce Job
MapReduce Tasks
MapReduce (SpatialRecordReader, SpatialFileSplitter)
Index Information
Master
Slaves
File Data
Storage (Grid File, R-tree, R+-tree, kd-tree, Quadtree)

Fig. 1. SpatialHadoop system architecture [12].

SpatialHadoop, see in Figure 1 its architecture, is a comprehensive extension to Hadoop that injects spatial data awareness in each Hadoop layer, namely, the language, storage, MapReduce, and operations layers. In the *Language* layer, SpatialHadoop adds a simple and expressive high level language for spatial data types and operations. In the *Storage* layer, SpatialHadoop adapts traditional spatial index structures as Grid, R-tree and R$^+$-tree, to form a two-level spatial index [27]. SpatialHadoop enriches the *MapReduce* layer by new components to implement efficient and scalable spatial data processing. In the *Operations* layer, SpatialHadoop is also equipped with a several spatial operations, including range query, KNNQ and spatial join. Other computational geometry algorithms (e.g. polygon union, skyline, convex hull, farthest pair, and closest pair) are also implemented following a similar approach [27]. Moreover, in this context, [13] improved the processing of skyline query implemented in SpatialHadoop. Finally, we emphasize that our contribution (εDRQs as spatial operations) is located in the *Operations* layer, as we can observe in Figure 1 in the highlighted box.

Since our main objective is to include the *DRQs* into SpatialHadoop, we are interested in the MapReduce and operations layers. *MapReduce layer* is the query processing layer that runs MapReduce programs, taking into account that SpatialHadoop supports spatially indexed input files. And the *operation layer* enables the efficient implementation of spatial operations, considering the combination of the spatial indexing in the storage layer with the new spatial functionality in the MapReduce layer. In general, a spatial query processing in SpatialHadoop consists of four steps: (1) *Partitioning*, where the data is partitioned according to a specific spatial index. (2) *Pruning*, when the query is issued, where the master node examines all partitions and prunes those ones that are guaranteed not to include any possible result of the spatial query. (3) *Local spatial query processing*, where a local spatial query processing is performed on each non-pruned partition in parallel on different machines. And finally, (4) *Global processing*, where a single machine collects all results from all machines in the previous step and compute the final result of the concerned query. And we are going to follow this query processing schema to include the *DRQs* (εDRQ and $\varepsilon DRJQ$) into SpatialHadoop.

4 *DRQ* Algorithms in SpatialHadoop

4.1 εDRQ Algorithm in SpatialHadoop

In this section, we describe our approach to the εDRQ algorithm on top of SpatialHadoop. In general, our solution is similar to how *range query* algorithm [12] is performed in SpatialHadoop, except instead of having a rectangular region (window), now we have a circular region defined by the query point and the range of distances $[\varepsilon_1, \varepsilon_2]$.

Firstly, we can use these ε_1, ε_2 values in combination with the features of indexing that are provided by SpatialHadoop to further enhance the pruning phase. Before the *Map* phase begins, we exploit the indexes to prune cells that cannot contribute to the final result. We partition the point dataset by some method (e.g. Grid or Str [27]) into blocks of points. CELLSFILTER algorithm takes as input each block / cell in which the input set of points is partitioned. Using SpatialHadoop built-in function *minDistance* we can calculate the minimum distance between a cell and the query point. That is, if we find a block with points which cannot have a distance value smaller than ε_2, we can prune that block. Furthermore, we can use *maxDistance* to prune blocks with points which have distance values smaller than ε_1.

Once the filter is done, the answer is refined in the *Map* phase. In order to do this, a full-scan Algorithm 1 is implemented, where the set of points P is scanned giving points which satisfy the distance value restrictions ε_1 and ε_2 over the query point q as solutions. Notice that in the *Reduce* phase the results are just combined.

Algorithm 1 εDRQ MapReduce Algorithm

```
1: function MAP($P$: set of points, $q$: query point, $\varepsilon_1$: lb distance, $\varepsilon_2$: ub distance)
2:     for all $point \in P$ do
3:         $distance \leftarrow$ DISTANCE($point, q$)
4:         if $distance \leq \varepsilon_2$ then
5:             if $distance \geq \varepsilon_1$ then
6:                 OUTPUT(null, point)
7:             end if
8:         end if
9:     end for
10: end function
```

4.2 *εDRJQ* Algorithm in SpatialHadoop

In this section, we describe our approach to the $\varepsilon DRJQ$ algorithm on top of SpatialHadoop. This can be described as a special case of the KCPQ MapReduce job [6], where only we select the pairs of points in the range of distances $[\varepsilon_1, \varepsilon_2]$ for the final result. And the distance threshold will be always ε_2 instead of the distance of the K-th closest pair found so far [28]. Moreover, our approach adapts the two distance-based plane-sweep KCPQ algorithms (Classic and Reverse Run) for main-memory resident datasets [28] to $\varepsilon DRJQ$.

In [28], the *Classic Plane-Sweep* for KCPQ was reviewed and two new improvements were also presented, when the point datasets reside in main memory. In general, if we assume that the two point sets are P and Q, the *Classic* plane-sweep algorithm consists of the two following steps: (1) sorting the entries of the two point sets, based on the coordinates of one of the axes (e.g. X) in increasing order, and (2) combine one point (*pivot*) of one set with all the points of the other set satisfying $point.x - pivot.x \leq \varepsilon_2$. The algorithm chooses the *pivot* in P or Q, following the order on the sweeping axis.

In [28], a new distance-based plane-sweep algorithm for KCPQ was proposed for *minimizing the number of distance computations*. It was called *Reverse Run* plane-sweep algorithm. It also follows the *sort-and-scan paradigm* and it is based on two concepts. First, every point that is used as a *reference* point forms a *run* with other subsequent points of the same set. A *run* is a continuous sequence of points of the same set that doesn't contain any point from the other set. And second, the *reference* points (and their *runs*) are processed in ascending X-order (the sets are X-sorted before the application of the algorithm). Each point of the *active run* is compared with the points of the other set (*comparison* points) in the opposite or reverse order (descending X-order) and the pair of points is selected if its distance is smaller than or equal to ε_2.

The two improvements presented in [28], called *sliding window* and *sliding semi-circle*, can be applied both *Classic* and *Reverse Run* plane-sweep algorithms. For the *sliding window*, the general idea consists of restricting the search space to the closest points inside the window with width ε_2 and a height $2 * \varepsilon_2$ (i.e. $[0, \varepsilon_2]$ in the X-axis and $[-\varepsilon_2, \varepsilon_2]$ in the Y-axis, from the *pivot* or the *reference* point). And for the *sliding semi-circle* improvement, it consists of trying to reduce even more the search space, since we can only select those points inside the semi-circle centered in the *pivot* or in the *reference* point with radius ε_2.

Algorithm 2 εDRJQ MapReduce Algorithm

```
1: function MAP($P$: set of points, $Q$: set of points, $\varepsilon_1$: lb distance, $\varepsilon_2$: ub distance)
2:     SORTX($P$)
3:     SORTX($Q$)
4:     $Results \leftarrow$ EDRJQ($P, Q, \varepsilon_1, \varepsilon_2$)
5:     for all $DistanceAndPair \in Results$ do
6:         OUTPUT($DistanceAndPair.p, DistanceAndPair$)
7:     end for
8: end function

9: function CELLSFILTER($C$: set of cells, $D$: set of cells, $\varepsilon_1, \varepsilon_2$: lb and ub distances)
10:    for all $c \in C$ do
11:        for all $d \in D$ do
12:            $minDistance \leftarrow$ MINDISTANCE($c, d$)
13:            if $minDistance \leq \varepsilon_2$ then
14:                $maxDistance \leftarrow$ MAXDISTANCE($c, d$)
15:                if $maxDistance \geq \varepsilon_1$ then
16:                    OUTPUT($c, d$)
17:                end if
18:            end if
19:        end for
20:    end for
21: end function
```

The method for the εDRJQ in MapReduce is to adopt the *Map* phase join MapReduce methodology. The basic idea is to have P and Q partitioned by some method (e.g. Grid or Str [27]) into n and m blocks of points, respectively. Then, every possible pair of blocks (one from P and one from Q) is sent as the input for the *Filter* phase. In the same way as it was done in 4.1 a CELLSFILTER function prunes pairs of blocks which have maximum and minimum distances between them that do not match ε_1 and ε_2 values.

On the *Map* phase each *mapper* reads its pair of blocks and performs a εDRJQ algorithm (*Classic* or *Reverse Run*) between the local P and Q in that pair. That is, it computes the εDRJQ between points in the local block of P and in the local block of Q using a εDRJQ plane-sweep algorithm. To do so, each *mapper* sorts the local P and Q blocks in one axis (e.g., X axis in ascending order) and then applies a particular *EDRJQ* algorithm. The results from all *mappers* are just combined in the *Reduce* phase and written into HDFS files, storing only the pairs of points with distance in $[\varepsilon_1, \varepsilon_2]$, as we can see in Algorithm 2.

5 Experimentation

In this section we present the results of our experimental evaluation. We have used synthetic (Uniform) and real 2d point datasets to test our *DRQs* algorithms in SpatialHadoop. For synthetic datasets we have generated several files of different sizes using SpatialHadoop built-in uniform generator [12]. For real datasets we have used three datasets from OpenStreetMap[2] [12]: *BUILDINGS* which contains 115M points of buildings, *LAKES* which contains 8.4M points of water areas, and *PARKS* which contains 10M points of parks and green areas [12]. To study the scalability of the datasets, subsets have been created with sample ratios of 25%, 50% and 75% from the original dataset (100%). We have used two performance metrics, the execution time and the number of complete distance computations. All experiments are conducted on a cluster of 20 nodes on an OpenStack environment. Each node has 1 vCPU with 2GB of main memory running Linux operating systems and Hadoop 1.2.1.

ε Distance Range Queries (εDRQ). For the different experiments, the *query point* is located at the center of the Minimum Bounding Rectangles (*MBRs*) to which the different datasets are partitioned by the SpatialHadoop indexing method utilized. For example, the MBR of the synthetic datasets is [(0,0)-(1000000,1000000)], and the query point is at (500000,500000).

Our first experiment is to examine the effect of the data set size. As we expected for uniform datasets, the results the execution time are almost uniform, due to the fact that the number of blocks that pass the filtering phase is less than the number of *map* tasks. However, for the experiments with real datasets, the execution time varies due to the partitioning performed on the data, since in a grid-based partitioning, the number of cells depends on the size of the data and more or less blocks due to a different partitioning obtained. For example, the number of points to consider for datasets sampled at 50% and 100% ratio are the same and are more than at 25% and 75%. And as we expected, the number of items returned by the query increases by the same percentage as data grows.

The second experiment studies the effect of different spatial partitioning techniques included in SpatialHadoop (Grid and Str [27]) and ε value ($\varepsilon_1 = 0$ and varying $\varepsilon_2 = \varepsilon$). As shown in Figure 2 left graph, the choice of a partitioning technique does not greatly affect the execution time showing a similar behavior. Using *Grid* partitioned files, the execution time increases linearly until almost

[2] Available at `http://spatialhadoop.cs.umn.edu/datasets.html`

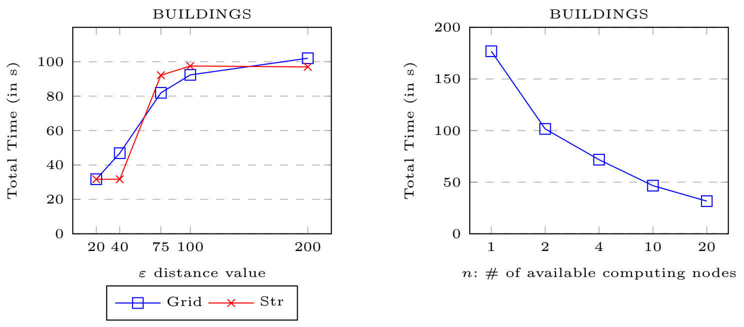

Fig. 2. Execution time vs. ε value (left) and execution time vs. n (right).

every point is selected and then it grows more slowly. As *Grid* partitioning is based on a uniform structure, the increment of ε values increases the number of selected blocks evenly. However, since *Str* partitioned files are nonuniform, the result is a stepped graph. For example, when ε value is 75, more blocks are selected and the execution time increases abruptly.

The third experiment aims to measure the speedup of the εDRQ algorithms, varying the number of computing nodes (n). Figure 2 right graph shows the impact of different numbers of computing nodes on the performance of parallel εDRQ algorithm. From this figure, it could be concluded that the performance of our approach has direct relationship with the number of computing nodes. It could be deduced that better performance would be obtained if more computing nodes are added. But, when the number of computing nodes exceeds the number of *map* tasks no improvement for that individual job is obtained.

ε Distance Range Join Queries (εDRJQ). The first experiment studies the effect of different spatial partitioning techniques included in SpatialHadoop (Grid and Str [27]). As shown in Figure 3 left graph, the choice of a partitioning technique greatly affects the execution time showing improvements of about 50% when using *Str* instead of *Grid*. Using *Grid* partitioned files, we get 175 combinations of blocks from input datasets, while using *Str* partitioned files just 79 combinations are obtained. This experiment is also useful to measure the scalability of the εDRJQ algorithms, varying the dataset sizes. We can observe that the execution time of our approach increases linearly.

The second experiment aims to find which of the different distance-based plane-sweep εDRJQ algorithms has the best performance. The execution times obtained do not show significant improvements between the different algorithms. This is due to various factors such as reading disk speed, network delays, the time for each individual task, etc. The metric shown in Figure 3 right graph is based on the number of times the algorithm performs a full calculation of the distance between two points. As shown in the right graph, any improvement (sliding Window or sliding Semi-Circle) on the *Classic* or *Reverse Run* algorithm obtains a much smaller number of calculations. The difference between these is not very noticeable being the *Semi-Circle Reverse Run* algorithm the one with better results in most of the cases.

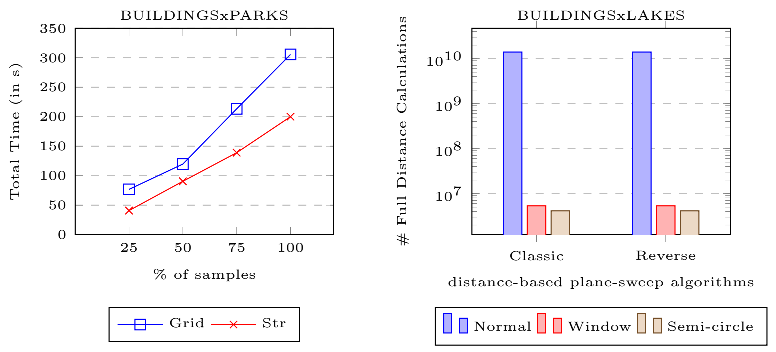

Fig. 3. Execution time vs. spatial partitioning techniques (left) and Number of complete distance computation vs. εDRJQ algorithm (right).

The third experiment studies the effect of the increasing of the ε value ($\varepsilon_1 = 0$ and varying $\varepsilon_2 = \varepsilon$). The scale of these ε values is different to that of the εDRQ (see Figure 2 left graph) due to the nature of the query. As show on Figure 4 left graph, the total execution time grows almost linearly as the ε value increases. It could be concluded that there is no real impact on the execution time but it must be taken into account that a higher ε value, the greater the possibility that pairs of blocks are not pruned and more *map* tasks could be needed.

The fourth experiment aims to measure the speedup of the εDRJQ algorithms, varying the number of computing nodes (n). Figure 4 right graph shows the impact of different node numbers on the performance of parallel εDRJQ algorithm, a behaviour similar to that of the εDRQ algorithm.

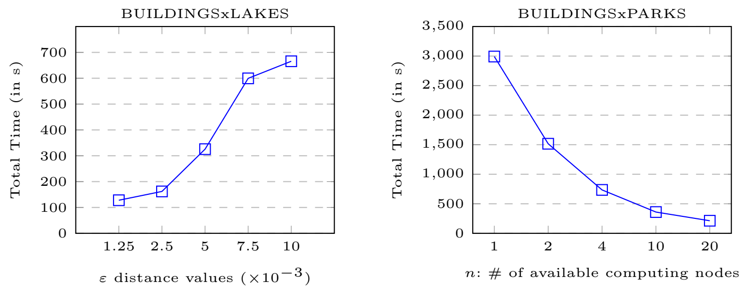

Fig. 4. Execution time vs. ε value (left) and execution time vs. n (right).

Conclusions from the experiments

– We have demonstrated experimentally the efficiency (in terms of total execution time and number of distance computations) and the scalability (in

terms of ε values, sizes of datasets and number of computing nodes) of the proposed parallel εDRQ and εDRJQ algorithms.
- Both plane-sweep-based algorithms (*Classic* and *Reverse Run*) used in the MapReduce implementation have similar performance in terms of execution time, although the *Semi-Circle Reverse Run* algorithm reduces slightly the number of complete distance computations.
- The use of a spatial partitioning technique included in SpatialHadoop [27] as *Str* (instead of *Grid*) improves notably the efficiency of the parallel εDRJQ algorithm. This is due to the fact that this variant organizes all partitions according to an R-tree structure at root level.

6 Conclusions and Future Work

DRQs (εDRQ and εDRJQ) are operations widely adopted by many spatial and GIS applications. They are characterized by a *filter* using a *distance range* over one or two datasets. These spatial queries have been actively studied in centralized environments, however, for parallel and distributed frameworks they have not attracted similar attention. For this reason, in this paper, we studied the problem of answering the DRQs in SpatialHadoop, an extension of Hadoop that supports spatial operations efficiently. To do this, we have proposed new parallel algorithms in MapReduce for the εDRQ and εDRJQ on big spatial datasets, adopting the plane-sweep methodology. We have also improved these MapReduce algorithms with the features of indexing that SpatialHadoop provides to further enhance the pruning phase. The performance of the algorithms in various scenarios with big synthetic and real-world points datasets has been also evaluated. And, the execution of such experiments has demonstrated the efficiency (in terms of total execution time and number of distance computations) and scalability (in terms of ε values, sizes of datasets and number of computing nodes) of our proposal. Future work might cover studying of *DRQs* with other partitioning techniques not included in SpatialHadoop and conducting tests to check the performance of the solution on clusters using the lastest version of Hadoop that can take advantage of new technologies such as YARN.

References

1. S. Shekhar and S. Chawla: "*Spatial databases - a tour*", Prentice Hall, 2003.
2. S. Zhang, J, Han, Z. Liu, K. Wang and Z. Xu: "SJMR: Parallelizing spatial join with MapReduce on clusters", *CLUSTER Conf.*, pp. 1-8, 2009.
3. S. You, J. Zhang and L. Gruenwald: "Spatial join query processing in cloud: Analyzing design choices and performance comparisons", *ICPP Conf.*, pp. 90-97, 2015.
4. C. Zhang, F. Li and J. Jestes: "Efficient parallel k-NN joins for large data in MapReduce", *EDBT Conf.*, pp. 38-49, 2012.
5. W. Lu, Y. Shen, S. Chen and B.C. Ooi: "Efficient processing of k nearest neighbor joins using MapReduce", *PVLDB* 5(10): 1016-1027, 2012.

6. F. García-García, A. Corral, L. Iribarne, M. Vassilakopoulos and Y. Manolopoulos: "Enhancing SpatialHadoop with Closest Pair Queries", *ADBIS Conf.*, *In press*, 2016.
7. Y. Kim and K. Shim: "Parallel top-K similarity join algorithms using MapReduce", *ICDE Conf.*, pp. 510-521, 2012.
8. Y.N. Silva and J.M. Reed: "Exploiting MapReduce-based similarity joins", *SIGMOD Conf.*, pp. 693-696, 2012.
9. J. Dean and S. Ghemawat: "MapReduce: Simplified data processing on large clusters", *OSDI Conf.*, pp. 137-150, 2004.
10. F. Li, B.C. Ooi, M.T. Özsu and S. Wu: "Distributed data management using MapReduce", *ACM Comput. Surv.* 46(3): 31:1-31:42, 2014.
11. C. Doulkeridis and K. Nørvåg: "A survey of large-scale analytical query processing in MapReduce", *VLDB J.* 23(3): 355-380, 2014.
12. A. Eldawy and M.F. Mokbel: "SpatialHadoop: A MapReduce framework for spatial data", *ICDE Conf.*, pp. 1352-1363, 2015.
13. D. Pertesis and C. Doulkeridis: "Efficient skyline query processing in SpatialHadoop", *Inf. Syst.* 54: 325-335, 2015.
14. J. Lu and R.H. Güting: "Parallel Secondo: Boosting database engines with Hadoop", *ICPADS Conf.*, pp. 738-743, 2012.
15. A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang and J.H. Saltz: "Hadoop-GIS: A high performance spatial data warehousing system over MapReduce", *PVLDB* 6(11): 1009-1020, 2013.
16. A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff and R. Murthy: "Hive - A warehousing solution over a MapReduce framework", *PVLDB* 2(2): 1626-1629, 2009.
17. S. You, J. Zhang and L. Gruenwald: "Large-scale spatial join query processing in cloud", *ICDE Workshops*, pp. 34-41, 2015.
18. J. Yu, J. Wu and M. Sarwat: "GeoSpark: A Cluster Computing Framework for Processing Large-Scale Spatial Data", SIGSPATIAL Conf., pp. 70-74, 2015.
19. Q. Ma, B. Yang, W. Qian and A. Zhou: "Query processing of massive trajectory data based on MapReduce", *CloudDB Conf.*, pp. 9-16, 2009.
20. S. Zhang, J. Han, Z. Liu, K. Wang and S. Feng: "Spatial queries evaluation with MapReduce", *GCC Conf.*, pp. 287-292, 2009.
21. A. Akdogan, U. Demiryurek, F.B. Kashani and C. Shahabi: "Voronoi-based geospatial query processing with MapReduce", *CloudCom Conf.*, pp. 9-16, 2010.
22. K. Wang, J. Han, B. Tu, J. Dai, W. Zhou and X. Song: "Accelerating spatial data processing with MapReduce", *ICPADS Conf.*, pp. 229-236, 2010.
23. J.M. Patel and D.J. DeWitt: "Partition based spatial-merge join", *SIGMOD Conf.*, pp. 259-270, 1996.
24. Y. Park, J.K. Min and K. Shim: "Parallel computation of skyline and reverse skyline queries using MapReduce", *PVLDB* 6(14): 2002-2013, 2013.
25. A. Eldawy, Y. Li, M.F. Mokbel and R. Janardan: "CG_Hadoop: computational geometry in MapReduce", *SIGSPATIAL Conf.*, pp. 284-293, 2013.
26. E.H. Jacox and H. Samet: "Metric space similarity joins", *ACM Trans. Database Syst.* 33(2): 1-38, 2008.
27. A. Eldawy, L. Alarabi and M.F. Mokbel: "Spatial partitioning techniques in SpatialHadoop", *PVLDB* 8(12): 1602-1613, 2015.
28. G. Roumelis, M. Vassilakopoulos, A. Corral and Y. Manolopoulos: "A new plane-sweep algorithm for the K-closest-pairs query", *SOFSEM Conf.*, pp. 478-490, 2014.